REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

2

	2. REPORT DATE	3. REPORT TYPE AND DATE COVERED
	August 1981	Technical Note--Oct 79-Aug 81

4. TITLE AND SUBTITLE
Project STEAMER: IV. A Primer on Conlan--A Constraint-Based Language for Describing the Operation of Complex Physical Devices

5. FUNDING NUMBERS
0603720N
Z1177-PN
Z1177-PN.03

6. AUTHOR(S)
Kenneth D. Forbus, Bolt Beranek & Newman, Inc.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Navy Personnel Research and Development Center
San Diego, California 92152-6800

8. PERFORMING ORGANIZATION REPORT NUMBER
NPRDC-TN-81-26

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Chief of Naval Operations (OP-01), Navy Department,
Washington, DC 20350-2000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The main objective of the STEAMER effort is to develop and evaluate advanced knowledge-based techniques for use in low-cost portable training systems. The project is focused on propulsion engineering as a domain in which to investigate these computer-based training techniques.

This report, the fourth in a series on the STEAMER project, describes a computer language that allows the description of a set of objects, the composition of these objects into complex system descriptions, and the simulation of the modelled system in a qualitative manner.

14. SUBJECT TERMS
Propulsion engineering; intelligent computer-assisted instruction; artificial intelligence; computer graphics

15. NUMBER OF PAGES
22

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

PRDC TN 81-26 AUGUST 1981

PROJECT STEAMER: IV. A PRIMER ON CONLAN--A CONSTRAINT-BASED LANGUAGE FOR DESCRIBING THE OPERATION OF COMPLEX PHYSICAL DEVICES

NAVY PERSONNEL RESEARCH
AND
DEVELOPMENT CENTER
San Diego, California 92152

NPRDC Technical Note 81-26 August 1981

PROJECT STEAMER: IV. A PRIMER ON CONLAN--A CONSTRAINT-BASED LANGUAGE FOR DESCRIBING THE OPERATION OF COMPLEX PHYSICAL DEVICES

Kenneth D. Forbus

Bolt Beranek and Newman, Inc.
Cambridge, MA 02138

Reviewed by
John D. Ford, Jr.

Released by
James F. Kelly, Jr.
Commanding Officer

Navy Personnel Research and Development Center
San Diego, California 92152

FOREWORD

This research and development was conducted under contract N00123-81-D0794 in support of Navy Decision Coordinating Paper Z1177-PN (Advanced Computer-Aided Instruction), subproject Z1177-PN.03 (STEAMER: Advanced Computer-Based Training for Propulsion and Problem Solving). It was sponsored by the Chief of Naval Operations (OP-01). The main objective of the STEAMER effort is to develop and evaluate advanced knowledge-based techniques for use in low-cost portable training systems. The project is focused on propulsion engineering as a domain in which to investigate these computer-based training techniques.

This report, the fourth in a series on the STEAMER project, describes a computer language that allows the description of a set of objects, the composition of these objects into complex system descriptions, and the simulation of the modelled system in a qualitative manner. Previous reports described an initial framework for developing techniques for automatically generating explanations of how to operate complex physical devices, provided a user's manual for the STEAMER interactive graphics package, and described a method for generating explanations using qualitative simulation (NPRDC TNs 81-21, 81-22, and 81-25 respectively). Intended users of these reports are system maintainers and other research personnel.

Appreciation is extended to personnel at the Surface Warfare Officers School at Newport, Rhode Island and the Propulsion Engineering School, Great Lakes, who participated in several beneficial discussions about the nature of the training problem being addressed in this R&D effort.

The contracting officer's technical representative was Dr. James D. Hollan.

JAMES F. KELLY, JR.
Commanding Officer

JAMES J. REGAN
Technical Director

SUMMARY

Problem

The main objective of the STEAMER effort is to develop and evaluate advanced knowledge-based techniques for use in low-cost portable training systems. The project is focused on propulsion engineering as a domain in which to investigate these computer-based training techniques. One of the important problems of the STEAMER system is generating explanations about the operation of propulsion plant components and subsystems.

Objective

This report describes CONLAN, a constraint-based programming language well suited for describing and analyzing complex devices. The descriptions can be used to generate understandable explanations and animate diagrams to explain the operation of complex devices.

Results

CONLAN is a computer language based on the idea that a system to be modelled is best described by specifying the constraints on the relationship of various parameters. An example of such a constraint is the ideal gas law stating that pressure in a container is proportional to temperature divided by volume. CONLAN enables the description of a set of objects and the composition of these objects to describe complex systems such as reducing valves and boiler control systems. Once described, the system can be simulated qualitatively.

CONLAN has been used in STEAMER as a basis for generating coherent understandable explanations of the operation of propulsion plant devices from a qualitative simulation of the device operation. These constraint-based, qualitative simulation techniques make possible learning environments in which students can experiment with complex devices and see explanations of the effects of various changes.

CONTENTS

	Page
INTRODUCTION	1
Problem	1
Objective	1
RESULTS	1
Overview of CONLAN	2
Specifying Constraints	5
Referencing	5
Prototypes	5
Complex Parts and Prototypes	9
Building Constraint Networks	10
Running the Interpreter	10
REFERENCES	15
APPENDIX--GLOSSARY OF FLAGS AND FUNCTIONS	A-0

LIST OF FIGURES

1. A simple constraint network: A three-input adder . . . 3
2. An adder constraint with rules . . . 4
3. Prototype for a multiplier constraint . . . 7
4. Same-level constraints with wiring rules . . . 7
5. A three-input adder made by sharing parts . . . 8
6. Chamber prototype . . . 11
7. Building a network for temperature conversion . . . 13
8. Qualitative reasoning about properties of a gas . . . 14

INTRODUCTION

Problem

The main objective of the STEAMER effort is to develop and evaluate advanced knowledge-based techniques for use in low-cost portable training systems. The project is focused on propulsion engineering as a domain in which to investigate these computer-based training techniques. One of the important problems of the STEAMER system is generating explanations about the operation of propulsion plant components and subsystems.

Objective

This report describes CONLAN, a constraint-based programming language well suited for describing and analyzing complex devices. The descriptions can be used to generate understandable explanations and animate diagrams to explain the operation of complex devices.

RESULTS

Complex systems are often modelled by describing their parts and the relationships between these parts. These relationships are often expressed in mathematics as constraint equations. Unlike normal computer languages, a constraint expression does not include a prescription for its use. The statement "Z = X + Y," for example, is just as much about how to compute X given Y and Z as how to compute Z given X and Y.

Recently attention has been focused on making programs work in the same way. CONLAN is such a constraint language. It was originally developed as a pedagogic tool to describe in a general way the technique of propagation of constraints used in EL (Stallman & Sussman, 1977) and other programs. The original version of CONLAN is described in Steele and Sussman (1978), who provide a clear and concise exposition of the main ideas of constraint languages.

While some of the syntax is the same, the language described here under the same name is a serious working tool. Its power has been extended in several ways--many

"hooks" have been added to enable it to be integrated into larger systems, and the implementation is designed with efficiency in mind. It has been used in a program for understanding motion (Forbus, 1980), generating English explanations of feedback devices (Forbus & Stevens, 1981), conducting musical composition experiments, developing simple qualitative physiological models, and implementing a fledgling "Naive Physics" describing the processes in a steam plant.

Overview of CONLAN

The basic object in CONLAN is a constraint; that is, a structured description with parameters. Constraints are created by instantiating prototypes and can be joined together into networks to describe complex systems or situations.

Constraints are made up of cells and parts. Cells hold the values of the parameters of the description. For example, a constraint describing a ball might include cells to hold the X and Y coordinates of its position at some instant, a value constraint might include a cell whose value indicates whether that valve is open or closed, and an inequality constraint would include cells naming the quantities being related.

Parts of a constraint are themselves simpler constraints. For example, the ball constraint mentioned above might contain a vector constraint to represent its velocity. An inequality constraint would include a "Taxonomy" constraint that would insist that exactly one of the possible relations Greater-Than, Less-Than, or Equal-To holds between the quantities being compared. For a system of any complexity, there are typically several levels of such embedding.

The cells and parts in a constraint are connected by sharing structure. For example, in a real physical system, two pipes might be joined by having an end of one pipe welded to the end of the other pipe, thus sharing a port between the two. When defining a constraint prototype, a similar technique is used. To connect constraints into a network that describes a complex situation, a mechanism that gives the effect of shared parts without actually modifying the structures is used. This allows the constraints in a network to be easily disconnected when required.

A convenient notation for constraint networks is to draw constraints as objects and cells as terminals, as in logic diagrams. Shared parts can be denoted by "wiring" two parts together. These conventions are illustrated in Figure 1, where two adders have been hooked up to form a three-input adder.

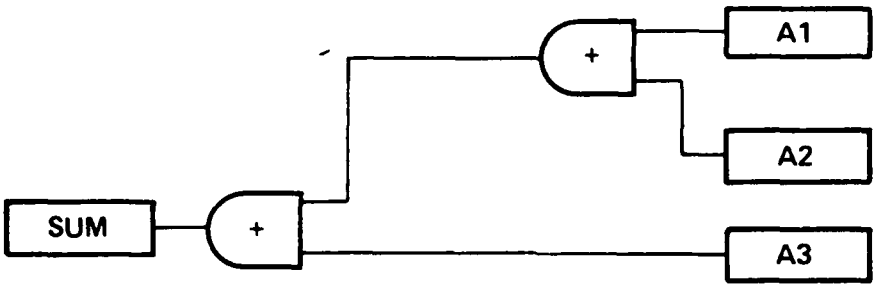

Figure 1. A simple constraint network: A three-input adder.

Relationships between the parameters in a constraint are enforced by a set of rules. Each rule has some set of cells that must be known before it can be run (called the uses set) and a cell that it will set if it runs successfully, called the set cell. The way to think of these rules is as follows: "If one know the uses values, then one can compute the set values." Rules can also return a special value that indicates an inability to come up with a value for a particular set of input values, and to signal a complaint when an inconsistency is discovered.

Computation within CONLAN is done by forward deduction as follows: When a cell is given a value, each rule that uses it (the users) is examined to see if all cells in its uses set are known. If they are, the rule is placed on a queue and eventually run. A rule may return a value, indicate that it has failed to get a result, or signal a contradiction. If a value is returned, it is placed in the set cell. If a contradiction occurs, a user procedure is

invoked to analyze the difficulty. When a cell is set, a note is made of the rule that set it (called the informant) so that the reasons for each value may be determined. Setting some new cell may in turn cause other rules to be queued, and the process continues until no more rules remain on the queue.

Using the diagram convention above, rules are represented by circles. An arrow into a circle denotes that the cell it comes from is used by the rule, and the single arrow out of the circle points at the cell that could be set by that rule. A two-input adder, for example, would contain three rules corresponding to the three ways it might be used. These are illustrated in Figure 2.

```
(Constraint Adder    ((A1 cell)
                      (A2 cell)
                      (Sum cell))

(formulae
 (Sum (A1 A2)  (+ A1 A2))
 (A1 (Sum A2)  (- Sum A2))
 (A2 (Sum A2)  (- Sum A1))))
```

Figure 2. An adder constraint with rules.

The advantage of the hierarchical structure of constraints is that the rules attached to the simpler constraints work through shared cells to enforce the semantics of the larger description. Thus, instead of writing all of the rules necessary to build a three-input adder, one can simply "wire up" two two-input adders.

A special class of rules and cells is provided to facilitate the construction of complex constraint networks. An indirect cell is one that holds another constraint, as opposed to a value. References to parts that include this kind of cell can be made to act as if their value were an actual part of the original constraint. Wiring rules are run when all of the indirect cells they depend on are filled and specify which parts are to be connected to each other and to the constraint of which the rules are a part. Making these connections can result in new values being deduced, since some cells may now take on values. The

dependence of values on these connections is stored so that, if the connection is broken, the network will remain in a consistent state.

Specifying Constraints

This section describes the syntax of constraint prototypes and how to write them. CONLAN is implemented in Lisp and that is reflected in the syntax. (For an introduction to Lisp and its syntactic conventions, see Winston (1977) or Winston and Horn (1980).

Referencing

A uniform convention is used for referring to parts of a structure both within a prototype description and for parts of a constraint network. For example, a list of the form:

(>> isolation bypass main-steam-stop)

should be read as "The isolation of the bypass of the main-steam-stop." The depth of such a reference can be arbitrary. If the values of indirect cells are being accessed, ">>i" should be used instead.

Prototypes

A prototype is specified by a list whose first element is the keyword "CONSTRAINT" and whose second element is a list of parts.[1] Each element of the parts list specifies the name of the part and the kind of thing it is. For example, an adder has three parts (A1, A2, and SUM), each of which is a cell.

The rest of the prototype specification is some number of a set of special forms. Formulae describes the rules that relate the cells of the constraint. Wiring describes the interconnections that need to be made when indirects are known. If-Removed is used for updating external representations. R== is used to specify shared parts. Each will be described in turn.

1. Formulae. The formulae statement consists of a list of rules. Each rule is of the form:

---

[1]The keyword "DEFBODY" is also provided if using a TAGS package is desired.

(<set> <uses> <body>)

or

<name> <set> <uses> <body>)

where <set> is the cell the rule is intended to supply a value for, <uses> are the other cells in the constraint it uses, and <body> is a lisp expression in terms of the <uses> names whose value is the result of the rule. The <name> form of the rule specification is mainly provided for mnemonic value. It sometimes is useful when an external system runs and analyzes the constraint net and must know how a value came about.

Two potential rule values are treated specially. The atom *DISMISS* means the rule was unable to compute a value with the specific values of its input parameters, and so none is assigned. The atom *LOSE* means the rule has detected an inconsistency, and a contradiction should be signaled. If <set> is nil, then whatever value (except *LOSE*) will be ignored. This convention is useful for rules that run for effect, such as manipulating an external representation or looking for inconsistent situations. The formulae statement is illustrated by the description of the multiplier constraint in Figure 3.

2. Wiring Rules. The wiring statement specifies how the values of indirect cells should be connected once they are known. Each rule in the statement has the syntax:

(<uses> <form-1> <form-2> . . . <form-n>)

The <uses> list contains both indirect cells and the names of parts of the constraint being described that will be referred to within the rule. The rule is run whenever all the indirect cells have values. The <form-i> are either equality statements between parts or assignments of values to cells. To enable referencing the constraint from within its rules, the atom $SELF always has the value of the constraint the rule is from when the rule is run. While arbitrary Lisp code could be placed inside the wiring rule, equality statements and assignments of values to cells are the only types whose consequences are automatically retacted when the connection is broken (by forgetting the value of an indirect cell). Figure 4 illustrates wiring rules.

```
(defbody multiplier ((m1 cell)
                     (m2 cell)
                     (product cell))
 (formulae
  (product (m1) (cond ((NEARLY-ZERO? m1) 0.0)
                      (t *dismiss*)))
  (product (m2) (cond ((NEARLY-ZERO? m2) 0.0)
                      (t *dismiss*)))
  (product (m1 m2)(COND ((NOT (OR (NEARLY-ZERO? M1)
                                  (NEARLY-ZERO?  M2)))
                         (times m1 m2))
                        (T *DISMISS*)))
  (m1 (product m2)(cond ((not (NEARLY-ZERO? m2))
                         (quotient product m2))
                        (t (cond ((NEARLY-ZERO? product)
                                  *dismiss*)
                                 (t *lose*)))))
  (m2 (product m1) (cond ((not (NEARLY-ZERO? m1))
                          (quotient product m1))
                         (t (cond ((NEARLY-ZERO? product)
                                   *dismiss*)
                                  (t *lose*)))))))
```

Figure 3. Prototype for a multiplier constraint.

```
(Constraint Same-Level-Numeric ((Container1 conlan-cell)
                                (Container2 conlan-cell)
                                (Level cell))
;;;Assumes level parameter of the containers it is given
;;;are numbers in a common global coordinate frame, and
;;;equates them - when one is known, the other is, and
;;;they must be consistent.
(Wiring ((Container1 Level) (== (>> Level Container1)
                                Level))
        ((Container2 Level) (== (>> Level Container2)
                                Level))))

(Constraint Same-Level-Symbolic ((Container1 conlan-cell)
                                 (Container2 conlan-cell)
                                 (Comp comparator))
;;;Assumes global frame, but uses the level cell as a
;;;symbolic parameter. The comparator constraint asserts
;;;the equality in a global lattice and thus may enable
;;;other comparators to draw conclusions.
(Wiring ((Container1 Comp)
         (Set-parameter (>> A comp)
                        (>> Level Container1)))
        ((Container2 Comp)
         (Set-parameter (>> B comp)
                        (>> Level Container2))))
;assert that whatever the levels are, they are equal.
(Constant (>> Equal-To? Comp) T))
```

Figure 4. Same-level constraints with wiring rules.

3. IF-REMOVED Statement. Sometimes it is necessary to use a constraint network with another kind of representation, such as a diagram or inequality expert. Since the body of a rule is a piece of Lisp code, it can perform bookkeeping by means of side effects. However, some way must exist to undo these effects when the values used in the rule are forgotten. The If-Removed statement allows this. The elements of If-Removed statements are called "forget functions," and have the syntax:

(<trigger> <form-1> <form-2> . . . <form>)

The trigger can be either a cell or a named rule. When the trigger is forgotten, the forms are evaluated from left to right. The environment for their evaluaiton includes bindings for (1) $SELF, the constraint it belongs to, (2) $SET-CELL, the cell being set, (3) $VALUE, the value in that cell, and (4) $INFORMANT, the name of the rule.

4. Sharing Structure. To specify shared structure in a prototype, the statement

(R== <ref-1> <ref-1> . . . <ref-n>)

is used.[2] Each reference must evaluate to a cell and, during instantiation of the prototype, only one thing will be created and given all of the names. Figure 5 shows a prototype for a 3-Adder that illustrates R==.

```
(Constraint three-adder      ((A1 cell)
                              (A2 cell)
                              (A3 cell)
                              (Sum cell)
                              (Add1 adder)
                              (Add2 adder))
(R== A1 (>> A1 Add1))
(R== A2 (>> A2 Add1))
(R== (>> Sum Add1) (>> A1 Add 2))
(R==A3 (>> A2 Add2))
(R== Sum (>> Sum Add2)))
```

Figure 5. A three-input adder made by sharing parts.

[2]As a mnemonic, think of this as "Really equal." If you are familiar with Lisp, think of it as "Rplac-equal."

Complex Parts and Prototypes

A constraint prototype normally has a fixed number of parts. This is not always convenient. For example, one would not want to define a separate ADDER constraint for each of the number of inputs desired. Two types of special prototypes are provided to remedy this.

The first type is the Macro prototype. The user must define a Lisp macro that, when called with a parameter, will generate a prototype with the appropriate number of parts. For example,

(AND005 (And-gate 4))

in the parts list of a constraint prototype says that AND005 will be a four-input AND gate. The macro call should either return the appropriate prototype, if it exists, or otherwise build one.

Macro prototypes are not always sufficient. Consider building a description of a container, including a description of the net flow of some fluid in the container. The number of ports determines the net flow, and each of the ports must be connected to the appropriate inputs of the net flow computation by sharing ports. A Dynamic prototype is provided for this.

A Dynamic prototype is specified by a PROTOTYPE-NAME-GENERATOR, a STATIC part, and a DYNAMIC part. The prototype name generator must parse the parameters of the constraint. In the example above, the call might be

(Chamber3 (Chamber Fluid-Ports 4. Heat-Ports 2.)),

which says to build a chamber from a prototype with four fluid ports and two heat ports.

The static part contains the prototype specifications that are the same for all choices of parameters. The dynamic part specifies the expansion parameters, what must be done for each value of them, and what parts exist in common for all. A laborious example is in Figure 6.

Building Constraint Networks

There are two ways to build a network out of individual constraints. The first way is to use indirect cells. The second is to equate parts. Equating parts makes the constraints behave as if they were the same thing. The syntax is:

(== <ref> <ref>)

Unlike R==, the parts referred to can be any two things of the same type rather than just cells. The equality mechanism actually works by making special equality constraints hold between corresponding cells of the parts. The equality constraints contain two rules, called "1<-2" and "2<-1," which place the value of one of the cells into the other as soon as it is known. The connection can be broken by using UN==.

Running the Interpreter

Two examples of interaction with CONLAN are provided in Figures 7 and 8. In these examples, ">>" is the interpreter prompt for more input, and indented lines are annotations. The environment was initialized each time by calling (Conlan-Init). The function call (Run-Constraints) starts the read-eval-propagate-print cycle. Figure 7 illustrates the construction and debugging of a network to convert temperatures, and Figure 8, the qualitative use of a gas law.

```
;;;Containers have no defined flow direction, but
;;do have a certain capacity.  The only way something
;;may enter or leave a container is by a port (either
;;fluid or thermal).  This model does not include
;;metric properties of the ports nor the effects of
;;gravity.
;;
;;A chamber can have some number of fluid and thermal
;;ports, therefore it is a "dynamic" constraint.

(static-part chamber
  (Parts  (top cell)(bottom cell)(sidel cell)
          (side2 cell)(capacity cell)
          ;;The above parts describe the geometry of
          ;;a chamber.
          (contents conlan-cell)
          (PC possible-contents)
          (level cell)
          (liquid-history chamber-fluid-history-computer)
          (steam-history chamber-fluid-history-computer)
          (air-history chamber-fluid-history-computer)
          (thermal-history
           chamber-thermal-history-computer)
          (component1 cell)
          (component2 cell)
          (mix-type cell))
  (wiring ((contents component1 component2 mix-type)
           (== component1 (>> component1 contents))
           (== component2 (>> component2 contents))
           (== mix-type (>> mixture-type contents))))
  (R== (>> net-flow steam-flow-computer)
       (>> net-flow steam-history))
  (R== (>> net-flow air-flow-computer)
       (>> net-flow air-history))
  (R== (>> net-flow liquid-flow-computer)
       (>> net-flow liquid-history))
  (R== (>> net-flow heat-flow-computer)
       (>> net-flow thermal-history)))
 ;;;Dynamic part of Chamber prototype
 ;;variables are denoted by "&&" prefix
 ;;fp=fluid ports
 ;;tp=thermal ports
```

Figure 6. Chamber prototype.

```
 (Dynamic-Part CHAMBER
 (VARS (fluid-ports . &&fp)
       (heat-ports . &&tp))
 (For-Each
  &&fp ;Make parts necessary for each fluid port
  (Parts ((&&fp . port) conlan-cell)
         ((&&fp . local-port) local-port)
         ((&&fp . MPT) Mixture-Touch-Computer))
  (Wiring (((&&fp . port) (&&fp . local-port))
           (set-parameter (>> port (&&fp . local-port))
                          (&&fp . port))))
  (Identities
   (R== (&&fp . port)
        (>> port (&&fp . local-port))
        (>> (&&fp . port) steam-flow-computer)
        (>> (&&fp . port) air-flow-computer)
        (>> (&&fp . port) liquid-flow-computer))
   (R== contents (>> contents (&&fp . MPT)))
   (set-parameter (>> port (&&fp . MPT))
                  (&&fp . local-port))
   (R== (>> air-flow-direction flow-senses
            (&&fp . local-port))
        (>> (&&fp . flow-direction)
            air-flow-computer))
   (R== (>> steam-flow-direction flow-senses
            (&&fp . local-port))
        (>> (&&fp . flow-direction)
            steam-flow-computer))
   (R== (>> liquid-flow-direction flow-senses
            (&&fp . local-port))
        (>> (&&fp . flow-direction)
            liquid-flow-computer))))

;;Rest of the dynamic part

(For-Each
 &&tp ;Make parts necessary for each thermal port
 (Parts ((&&tp . heat-port) conlan-cell)
        ((&&tp . local-heat-port) local-heat-port))
 (Identities
  (R== (&&tp . heat-port)
       (>> (&&tp . tport) heat-flow-computer))
  (R== (>> flow-sense (&&tp . local-heat-port))
       (>> (&&tp . flow-direction) heat-flow-computer))))
(In-Common
;Make shared parts with right number of terminals
 (Wiring ((contents pc)
          (== pc (>> pc contents))))
 (Parts (steam-flow-computer
         (net-flow-computer ports (&&limit &&fp)))
        (air-flow-computer
         (net-flow-computer ports (&&limit &&fp)))
        (liquid-flow-computer
         (net-flow-computer ports (&&limit &&fp)))
        (heat-flow-computer
         (net-flow-computer ports (&&limit &&tp))))))
```

Figure 6. (Continued).

```
        first start constraint interpreter

>(run-constraints)
>>(create Fahrenheit 'cell)
G0002
>>(create centigrade 'cell)
G0004

        create the parts to perform the computation

>>(create add1 'adder)
G0006
>>(create mul1 'multiplier)
G0011
>>(create mul2 'multiplier)
G0016

        Now we connect them up

>>(== Fahrenheit (>> sum add1))
IDENTITY
>>(set-parameter (>> a1 add1) 32.0)
32.0
>>(== (>> a2 add1) (>> product mul1))
IDENTITY
>>(== (>> m1 mul1) centigrade)
IDENTITY
>>(== (>> m2 mul1) (>> m1 mul2))
IDENTITY
>>(set-parameter (>> product mul2) 5.0)
5.0
>>(set-parameter (>> m2 mul2) 9.0)
9.0
>>(set-parameter centigrade 100.0)
100.0
>>(what-is Fahrenheit)
FAHRENHEIT = 87.555555
T

        Oops! We can use dependencies to find the bug...

>>(why Fahrenheit)
I used rule (1<-2 >> 1<=>2) on the following inputs:
   (>> SUM ADD1)
(G0009)
>>(premises Fahrenheit)
(>> M2 MUL2) = 9.0
(>> PRODUCT MUL2) = 5.0
(>> CENTIGRADE) = 100.0
(>> A1 ADD1) = 32.0
T

        Must switch the constants around-

>>(change-parameter (>> m2 mul2) 5.0)
5.0
>>(change-parameter (>> product mul2) 9.0)
9.0

        Notice that Fahrenheit is already recomputed-

>>(what-is Fahrenheit)
FAHRENHEIT = 212.0
T
>>(un== (>> m1 mul1) centigrade)
T

        removing a part can undo deductions-

>>(what-is Fahrenheit)
Sorry, I don't know it.
```

Figure 7. Building a network for temperature conversion.

```
>>(create boyle 'gas-law)
G0007

   The gas-law constraint encodes PV=KT in the IQ algebra

>>(partnames? boyle)
(PRESSURE TEMPERATURE VOLUME P1 P2)

   P1 and P2 are adders that work with the IQ algebra.

>>(set-parameter (>> temperature boyle) 'c)
C
>>(set-parameter (>> volume boyle) 'd)
D
>>(what-is (>> pressure boyle))
(>> (A1 P1 BOYLE) (PRESSURE BOYLE)) = U
NIL

   Temperature constant, volume decreasing means the
   pressure is increasing.

>>(why (>> pressure boyle))
I used rule (RULE-2 >> P1 BOYLE) on the following inputs:
   (>> (SUM P2 BOYLE) (SUM P1 BOYLE))
   (>> (A2 P1 BOYLE) (VOLUME BOYLE))
(G0012 G0010)

   Why cites the rule directly responsible.

>>(premises (>> pressure boyle))
(>> (A2 P1 BOYLE) (VOLUME BOYLE)) = D
(>> (A1 P2 BOYLE) (TEMPERATURE BOYLE)) = C
T

Premises are the assumptions the user made.

>>(results (>> temperature boyle))
Rule (RULE-1 . G0013) got (>> (SUM P2 BOYLE) (SUM P1 BOYLE)) = C
T
>>(results (>> sum p1 boyle))
Rule (RULE-2 . G0011) got (>> (A1 P1 BOYLE) (PRESSURE BOYLE)) = U
T

   Using RESULTS we can trace the path of the computation.

>>(change-parameter (>> volume boyle) 'u)
U

   Changing a parameter first forgets the results of the old
   value and then sets the new one.

>>(what-is (>> pressure boyle))
(>> (A1 P1 BOYLE) (PRESSURE BOYLE)) = D
NIL
>>(forget-parameter (>> temperature boyle))
((G0008 G0012 G0009) G0009)
>>(what-is (>> pressure boyle))
Sorry, I don't know it.  I need:
   (>> (SUM P2 BOYLE) (SUM P1 BOYLE))
 to use rule: (RULE-2 >> P1 BOYLE)
MUST-BE-ENTERED
>>
```

Figure 8. Qualitative reasoning about properties of a gas.

REFERENCES

Forbus, K. D., & Stevens, A. Project STEAMER: III. Using qualitative simulation to generate explanations of how to operate complex physical devices (NPRDC TN 81-25). San Diego: Navy Personnel Research and Development Center, August 1981.

Forbus, K. A study of qualitative and geometric knowledge in reasoning about motion. (Master's thesis). City: Massachusetts Institute of Technology, February, 1980.

Stallman, R. M. & Sussman, G. J. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence, 1977, 9, 135-196.

Stead, L. Project STEAMER: II. User's manual for the STEAMER interactive graphics package (NPRDC TN 81-22). San Diego: Navy Personnel Research and Development Center, August 1981.

Steele, G., & Saussman, G. Constraints. AI Memo 502, City: Massachusetts Institute of Technichology, Artificial Intelligence Laboratory, November 1978.

Stevens, A., & Steinberg, C. Project STEAMER: I. Taxonomy for generating explanations of how to operate complex physical devices (NPRDC TN 81-21). San Diego: Navy Personnel Research and Development Center, August 1981.

Winston, P., & Horn, B. Lisp. Reading, MA: Addison-Wesley, 1980.

Winson, P. Artificial intelligence. Reading, MA: Addison-Wesley 1977.

APPENDIX

GLOSSARY OF FLAGS AND FUNCTIONS

GLOSSARY OF FLAGS AND FUNCTIONS

This glossary contains the calls required to run the interpreter, create networks of constraints, and some relevant flags.

A.1 Flags

CONLAN-STATUS If interpreter ran out of things to do, QUIESCENT. If a clash occured, it should be noted by changing this variable to (CLASH <disputants>)

CONLAN-CONTRADICTION-HANDLER
If bound, it must be a function of one argument that will be called whenever a clash occurs. The argument is a list of two copies of a cell, with the conflicting values.

DEBUG-CONLAN If T, a message is printed when each cell is set. NIL muzzles it.

DISPUTANTS A list consisting of a cell and a cell-like entity which clash. This is the argument to the contradiction handling code.

STOP-CONLAN If T, the interpreter will stop.

A.2 Functions

(CONLAN-INIT) Initializes the interpreter state and destroys whatever networks currently exist.

(RUN-CONSTRAINTS)
Top level loop for system. Operates like the normal Lisp read-eval-print loop, except that the constraint interpreter is fired on each cycle and allowed to run until it is quiescent or has a clash.

(CREATE <name> <type>)
Creates a constraint of <type> called <name>. The name must be atomic and the type must be a legal constraint prototype.

(== <A> <B>) Equates <A> and <B>. In particular, whenever a cell in <A> gets a value the corresponding cell in <B> will get the same value.

(un== <A> <B>) Removes the equality between <A> and <B>, if any. It will not work on R=='s used in a prototype.

(SET-PARAMETER <cell> <value>)
Sets <cell> to <value>, giving the informant as the user. If an inconsistent value is given a clash occurs. If <value> is of the form (<number> <atom>) then <number> is assumed to be the value and <atom> the units.

(FORGET-PARAMETER <cell>)
If the user was the informant, the value for the cell is forgotten and all consequences of it forgotten as well. If the value did not come from the user nothing happens.

(CHANGE-PARAMETER <cell> <value>)
A combination of FORGET-PARAMETER and SET-PARAMETER.

(WHAT-IS <cell>)
Types the name and value of the cell if it is known, or what it would need to compute it if it is not.

(WHY <cell>) Types the informant of the cell and the values (if any) the rule used.

(PREMISES <cell>)
Types all of the user supplied parameters that were used in deducing the value for the cell.

(RESULTS <cell>)
Types what was directly computed using the value of the cell.

(SUPPLIERS <cell>)
Types the rules that could provide a value for that cell.

(USERS <cell>) Types the rules that use the cell.

(NEEDS <cell>) Types what is needed to compute a value for the cell from rules that directly set it.

(UNKNOWN-VALUES <constraint>)
Lists the cells in the constraint that do not yet have values.

(KNOWN-VALUES? <constraint>)
Lists the cells in the constraint that have values.

(INTERNAL-VALUES <constraint>)
Lists the cells that have been set by rules in the constraint.

(EXTERNAL-VALUES <constraint>)
Lists the cells that have been set by something outside the constraint.

(PRINT-RULE <rule>)
Pretty-prints the lisp code that comprises the rule.